Universidad San Carlos de Guatemala
Facultad de ingeniería.
Ingeniería en ciencias y sistemas

Título del Proyecto:

Golampi Interpreter

PONDERACIÓN: 45 pts

Tiempo estimado: 40 hrs

Índice

1. MARCO FORMATIVO..... 3
1.1. Valor..... 3
1.2. Competencia(s)..... 3
1.3. Objetivo SMART..... 3
2. Resumen Ejecutivo..... 4
3. Enunciado del Proyecto..... 4
3.1. Arquitectura de la aplicación..... 5
3.1.1. Propuestas..... 5
3.1.2. Diseño de la GUI..... 6
3.2. Generalidades del lenguaje..... 7
3.2.1. Identificadores..... 7
3.2.2. Comentarios..... 7
3.2.3. Tipos estáticos..... 8
3.3. Sintaxis del lenguaje..... 8
3.3.1. Bloques de sentencias..... 8
3.3.2. Variables..... 9
Declaración corta de variables:..... 9
3.3.3. Constantes..... 10
3.3.4. Valor nil..... 10
3.3.5. Operadores de asignación..... 10
3.3.6. Operadores Aritméticos..... 11
3.3.7. Operadores relacionales..... 13
3.3.8. Operadores lógicos..... 13
3.3.9. Sentencias de control de flujo..... 14
3.3.10. Sentencias de transferencia..... 16
3.3.11. Arreglos..... 17
3.3.12. Funciones y parámetros..... 19
3.3.13. Funciones embebidas..... 21
3.3.14. Función main..... 23
3.4. Reportes..... 23
3.4.1. Reporte de errores..... 23
3.4.2. Reporte tabla de símbolos..... 24
3.5. Alcance del proyecto..... 25
3.6. Entregables..... 25
4. Material de apoyo..... 26
5. Metodología..... 26
6. Recursos y herramientas a utilizar..... 26
7. Desarrollo de Habilidades Blandas..... 26
8. Cronograma..... 27

1. MARCO FORMATIVO

1.1. Valor

Nombre del valor	¿Cómo se aplica en tu laboratorio?
Pensamiento crítico	Analizando errores léxicos, sintácticos y semánticos para identificar su causa y proponer soluciones correctas dentro del diseño del lenguaje.

1.2. Competencia(s)

Tipo de Competencia	
Competencia General	El estudiante será capaz de diseñar e implementar un intérprete para un lenguaje con sintaxis similar a Golang, utilizando técnicas de análisis léxico, sintáctico y semántico, además de generar reportes asociados al código fuente procesado.
Competencia Específica	Desarrollar una interfaz gráfica funcional que permita al usuario crear, editar y ejecutar archivos, visualizar los resultados en consola y consultar los reportes generados durante el proceso de análisis y ejecución. Diseñar y construir la gramática del lenguaje Golampi utilizando ANTLRv4, permitiendo la generación automática de un analizador sintáctico conforme a la sintaxis definida del lenguaje. Implementar un sistema de análisis semántico basado en el recorrido del árbol sintáctico generado por ANTLRv4, orientado a la validación de estructuras del lenguaje y tipos de datos, así como a la generación de reportes de errores y la administración de una tabla de símbolos.

1.3. Objetivo SMART

Criterio	Objetivo
Qué (Específico)	Desarrollar un intérprete del lenguaje Golampi que incluya análisis léxico, sintáctico y semántico, ejecución desde la función main y generación de reportes de tabla de símbolos y errores mediante una interfaz gráfica.

Cuánto (Medible)	Lograr que el intérprete analice y ejecute correctamente al menos el 90% de un conjunto de casos de prueba predefinidos del lenguaje Golampi.
Cómo (Alcanzable)	Implementar el intérprete utilizando ANTLRv4 Parser para el análisis léxico y sintáctico, PHP como lenguaje de desarrollo y HTML/CSS para la interfaz gráfica.
Para qué (Realista)	Aplicar de forma práctica la teoría de compiladores y fortalecer competencias en el diseño e implementación de lenguajes de programación.
Cuándo (A tiempo)	Completar el desarrollo del proyecto dentro del cronograma establecido para el curso, cumpliendo los hitos definidos para cada fase.

2. Resumen Ejecutivo

El presente proyecto tiene como finalidad el desarrollo de un intérprete funcional para el lenguaje Golampi, un lenguaje académico inspirado en la sintaxis y semántica de Golang, orientado al aprendizaje práctico de la teoría de compiladores. La solución propuesta integra una arquitectura monolítica cliente/servidor con una interfaz gráfica que incorpora un editor de código, permitiendo la creación, edición y ejecución de programas escritos en Golampi.

El intérprete implementa las fases fundamentales del procesamiento de lenguajes: análisis léxico, sintáctico y semántico, apoyándose en el recorrido del árbol sintáctico generado por ANTLRv4 para la validación de estructuras del lenguaje, tipos de datos y contextos semánticos, así como para la gestión de la tabla de símbolos y la generación de reportes de errores. Asimismo, el sistema es capaz de localizar y ejecutar la función principal del programa, soportando estructuras de control, tipos de datos, expresiones, arreglos y funciones, conforme a las especificaciones del lenguaje definidas en el enunciado del proyecto.

Este proyecto busca fortalecer las competencias técnicas de los estudiantes en el diseño e implementación de intérpretes, promoviendo la aplicación de conceptos formales de compiladores, el uso de herramientas de generación de analizadores y buenas prácticas de ingeniería de software, dentro de un entorno académico controlado y evaluable.

3. Enunciado del Proyecto

3.1. Arquitectura de la aplicación

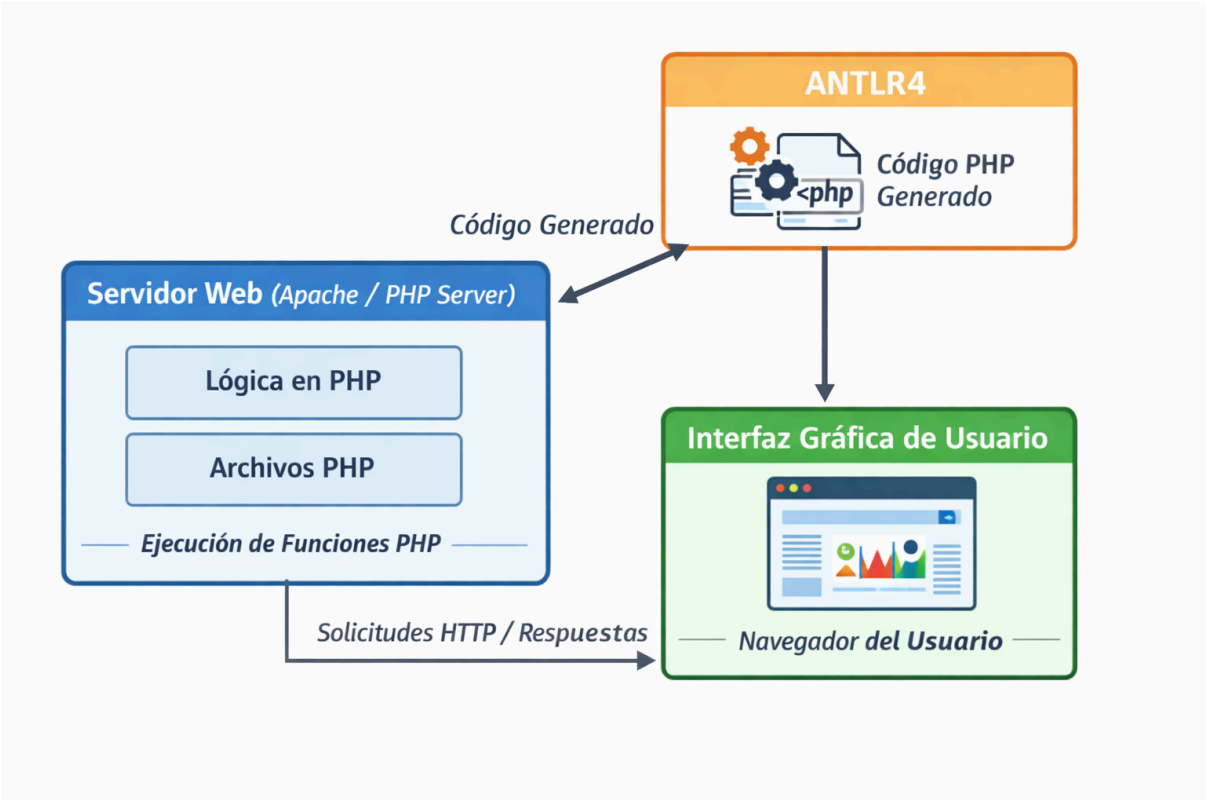
3.1.1. Propuestas

Aplicación Web Integrada (Monolítica)

Se define una arquitectura de aplicación web que funciona sobre un servidor web (Apache o un servidor ligero de PHP), donde se ejecuta la lógica principal en PHP. La comunicación HTTP se realiza únicamente entre el navegador del usuario y el servidor, ya que las interacciones internas del sistema se manejan mediante llamadas directas a funciones PHP, sin intercambio HTTP entre componentes internos.

El componente ANTLR4 se incluye a nivel arquitectónico para representar el origen del código PHP que forma parte del backend. Dicho código es previamente generado a partir de gramáticas definidas y desplegado en el servidor; durante la ejecución de la aplicación no se realiza ninguna generación dinámica de código, únicamente se ejecutan los archivos PHP ya existentes.

Finalmente, el sistema cuenta con un componente de interfaz gráfica de usuario, accesible desde el navegador, que permite al usuario interactuar con la aplicación. Esta interfaz envía solicitudes HTTP al servidor, el cual procesa las peticiones mediante sus funciones internas y devuelve las respuestas correspondientes, manteniendo una separación clara entre presentación, lógica y origen del código.



3.1.2. Diseño de la GUI

Barra de acciones:
La barra de acciones concentra las operaciones principales de interacción con el sistema y funciona como el punto de control de la GUI. Desde esta barra, el usuario puede crear o

limpiar el entorno de trabajo, cargar código fuente desde un archivo, guardar el contenido actual del editor y ejecutar o analizar el programa enviándolo al backend en PHP.

- Nuevo / Limpiar: Limpia el editor y la consola para iniciar una nueva prueba.
- Cargar archivo: Permite seleccionar un archivo de código y cargar su contenido en el editor.
- Guardar código: Descarga el contenido actual del editor como archivo de texto.
- Ejecutar / Analizar: Envía el código fuente al servidor PHP para su análisis o ejecución.
- Limpiar consola: Borra el contenido de la consola de salida.

Panel de edición de código:
Área central donde el usuario escribe o visualiza el programa fuente. Debe soportar texto multilínea y reflejar exactamente el contenido que será enviado al compilador. No realiza validaciones, únicamente sirve como entrada del sistema.

Consola de salida:
Panel destinado a mostrar la salida del intérprete, mensajes de ejecución y estados del proceso. Se limpia antes de cada ejecución y permite validar visualmente el comportamiento del programa.

Panel de reportes:
Zona dedicada a la descarga de artefactos generados por el compilador, habilitada después de una ejecución.

- Descargar resultado: Exporta la salida válida mostrada en la consola.
- Descargar errores: Genera un archivo con errores léxicos, sintácticos y semánticos.
- Descargar tabla de símbolos: Exporta la tabla generada durante el análisis semántico.

Comunicación Frontend y Backend:
La GUI actúa únicamente como intermediaria: envía el código fuente al servidor PHP y muestra las respuestas recibidas. Toda la lógica del intérprete/compilador, validaciones y generación de reportes reside exclusivamente en el backend.

Recomendaciones:
Para la GUI de Golampi se recomienda utilizar **HTML, CSS y JavaScript puro**, ya que estas tecnologías son suficientes para implementar el editor de código, la consola, los botones de acción y la descarga de reportes, manteniendo la interfaz simple y clara. El uso de frameworks como **React, Vue o Angular** no es necesario, pero queda a criterio del estudiante si desea utilizarlos, siempre que se respete la arquitectura y no se mezcle la lógica del intérprete con la interfaz.

Es importante aclarar que **la GUI es un requisito obligatorio del proyecto**: no se calificará una solución que únicamente consuma la API del compilador o que funcione solo por consola. La interfaz gráfica es necesaria para evidenciar la interacción completa con el intérprete, la visualización de resultados y la generación de reportes solicitados.

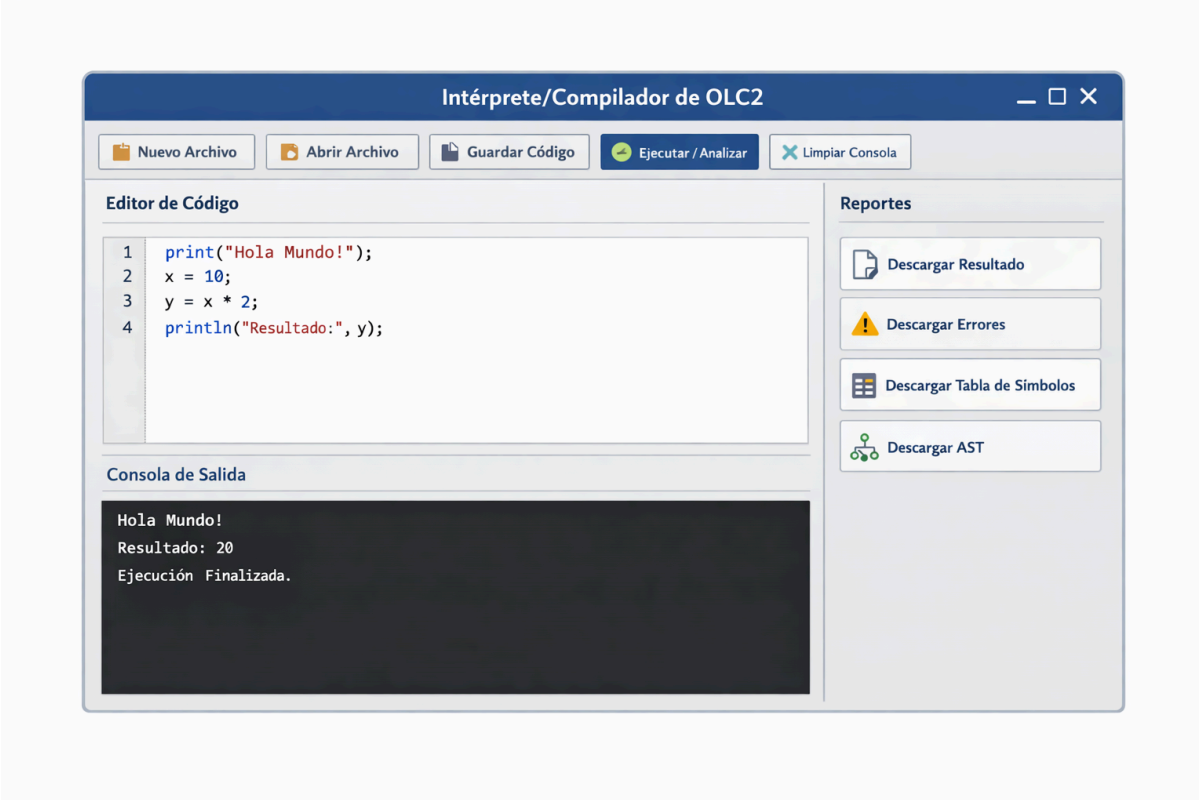


Figura de ejemplo

3.2. Generalidades del lenguaje

3.2.1. Identificadores

Un identificador es una secuencia de letras Unicode, dígitos y el carácter guión bajo (_). Debe comenzar obligatoriamente con una letra o con _, y los caracteres siguientes pueden ser letras, dígitos o _. No puede iniciar con un número, ni contener caracteres especiales como: '.', '$', ',', '-' u otros símbolos. Además, los identificadores son sensibles a mayúsculas y minúsculas y no pueden coincidir con palabras reservadas del lenguaje.

```
identifier  = letter { letter | digit } .
letter      = unicode_letter | "_" .
digit       = unicode_digit .
```

3.2.2. Comentarios

Los comentarios son fragmentos de texto ignorados por el compilador y se utilizan para documentar el código y mejorar su legibilidad. Pueden escribirse en una sola línea o en múltiples líneas y no afectan la ejecución del programa.

```
// Este es un comentario de una sola línea

/*
   Este es un comentario
   de múltiples líneas
*/
```

3.2.3. Tipos estáticos

Tipo	Descripción	Valor por defecto
`int32`	Entero con signo de 32 bits. Rango: -2^{31} a $2^{31} - 1$.	`0`
`float32`	Punto flotante de 32 bits (IEEE-754). Rango aproximado: 1.4E-45 a 3.4028235E38.	`0.0`
`bool`	Valor lógico: `true` o `false`.	`false`
`rune`	Carácter Unicode (alias de `int32`). Se escribe entre comillas simples.	`'\u0000'`
`string`	Cadena de texto Unicode. Se escribe entre comillas dobles.	`""`

3.3. Sintaxis del lenguaje

3.3.1. Bloques de sentencias

Un bloque de sentencias es una secuencia de declaraciones y sentencias encerrada entre llaves {}, que define un ámbito (scope) propio para las variables declaradas dentro de él. Los bloques se utilizan para agrupar instrucciones, controlar el flujo del programa y delimitar la visibilidad de identificadores.
Un bloque puede aparecer en funciones, estructuras de control (if, for, switch, select) y declaraciones anidadas. Las variables declaradas dentro de un bloque solo son accesibles dentro de ese bloque y de sus sub-bloques.

```
func main() {
      // Bloque de la función main
      x := 10

      if x > 5 {
      // Bloque del if
      y := x * 2
      fmt.Println(y)
      }

      // fmt.Println(y) // Error: y no es visible fuera del bloque if
}
```

3.3.2. Variables

El lenguaje permite la declaración y uso de variables para almacenar valores de distintos tipos primitivos. Cada variable debe declararse antes de ser utilizada y está asociada a un tipo estático, el cual se determina en tiempo de compilación. Las variables poseen un alcance léxico, limitado al bloque de sentencias en el que fueron declaradas.

Las variables pueden declararse mediante una sintaxis similar a Go, indicando explícitamente el tipo, y pueden inicializarse opcionalmente en el momento de la declaración. Si una variable no se inicializa, se le asigna el valor por defecto correspondiente a su tipo.

```
<varDcl> ::= <var> <id_list> <type>
         | <var> <id_list> <type> = <exp_list>
```

```
var x int = 10
var y int
var w, z int = 1, 2
x = 20

x = 20
// Ahora x vale 20
// y vale nil
// w vale 1 y z vale 2
// La longitud de la lista de ids y la de expresiones debe ser la misma.
```

Declaración corta de variables:

La declaración corta de variables permite declarar e inicializar una variable en una sola instrucción, sin especificar explícitamente su tipo. El tipo de la variable se infiere automáticamente a partir de la expresión del lado derecho.

Este tipo de declaración solo es válida dentro de bloques de sentencias (por ejemplo, dentro de funciones) y no puede utilizarse a nivel global. Además, al menos una de las variables del lado izquierdo debe ser nueva en el alcance actual.

```
<shortValDcl> ::= <id_list> := <exp_list>
```

```
func main() {
    x, y := 34, 68
    fmt.Println(x)
    fmt.Println(y)
}

// Output esperado:
```

```
// 34
// 68
```

3.3.3. Constantes

El lenguaje permite la declaración de constantes, las cuales representan valores inmutables que se determinan en tiempo de compilación. Una constante debe declararse antes de ser utilizada y está asociada a un tipo estático.

Las constantes se declaran utilizando la palabra reservada const, indicando un identificador, su tipo y un valor de inicialización. A diferencia de las variables, las constantes deben inicializarse obligatoriamente en el momento de su declaración, y su valor no puede modificarse posteriormente durante la ejecución del programa.

Las constantes poseen un alcance léxico, limitado al bloque en el que fueron declaradas, y pueden utilizarse en expresiones siempre que se respete la compatibilidad de tipos.

```
<consVarDcl> ::= <const> <id> <type> = <exp>
```

```
const max int = 100
const pi float32 = 3.14
```

3.3.4. Valor nil

En el lenguaje Golampi se utiliza la palabra reservada nil para hacer referencia a la nada, esto indicará la ausencia de valor, por lo tanto cualquier operación sobre nil será considerada un **error y dará nil** como resultado.

3.3.5. Operadores de asignación

Los operadores de asignación permiten asignar o actualizar el valor de una variable a partir de una expresión. El lenguaje soporta el operador de asignación simple y un conjunto limitado de operadores de asignación compuesta, los cuales combinan una operación aritmética o lógica con una asignación. El operando izquierdo debe ser una ubicación válida de almacenamiento, y el operando derecho debe ser una expresión compatible en tipo.

Operador	Descripción	Ejemplo
`=`	Asignación simple	`x = 5`
`+=`	Suma y asignación	`x += 2`
`-=`	Resta y asignación	`x -= 1`

*=	Multiplicación y asignación	x *= 3
/=	División y asignación	x /= 2

3.3.6. Operadores Aritméticos

Los operadores aritméticos permiten realizar operaciones matemáticas básicas sobre valores numéricos y, en ciertos casos, operaciones definidas sobre cadenas. El lenguaje soporta suma, resta, multiplicación, división, módulo y negación unaria.
Los operandos deben ser expresiones cuyos tipos sean compatibles con la operación, de otro caso el resultado dará **nil**. El tipo del resultado se determina en tiempo de compilación mediante reglas de promoción de tipos.

Suma:

+	int32	float32	bool	rune	string
int32	int32	float32		int32	
float32	float32	float32		float32	
bool					
rune	int32	float32		int32	
string					string

Resta:

-	int32	float32	bool	rune	string
int32	int32	float32		int32	
float32	float32	float32		float32	
bool					
rune	int32	float32		int32	
string					

Multiplicación:

*	int32	float32	bool	rune	string

int32	int32	float32		int32	string
float32	float32	float32		float32	
bool					
rune	int32	float32		int32	
string	string				

División:

/	**int32**	**float32**	**bool**	**rune**	**string**
int32	int32	float32		int32	
float32	float32	float32		float32	
bool					
rune	int32	float32		int32	
string					

Módulo:

%	**int32**	**float32**	**bool**	**rune**	**string**
int32	int32			int32	
float32					
bool					
rune	int32			int32	
string					

3.3.7. Operadores relacionales

Los operadores relacionales permiten comparar dos valores y producen como resultado un valor booleano (bool). Ambos operandos deben ser de tipos compatibles según las reglas del lenguaje. La validez de una comparación y su tipo resultante se determinan en tiempo de compilación.

Igualdad y Desigualdad:

== / !=	int32	float32	bool	rune	string
int32	bool	bool		bool	
float32	bool	bool		bool	
bool			bool		
rune	bool	bool		bool	
string					bool

Mayor, menor o igual:

> , >= , < , <=	int32	float32	bool	rune	string
int32	bool	bool		bool	
float32	bool	bool		bool	
bool					
rune	bool	bool		bool	
string					bool

3.3.8. Operadores lógicos

Los operadores lógicos permiten combinar o negar expresiones booleanas y producen como resultado un valor de tipo bool. Estos operadores solo están definidos para operandos de tipo bool. La validez de las operaciones se verifica en tiempo de compilación.

Restricción de cortocircuito

Los operadores lógicos binarios && (AND) y || (OR) deben evaluarse utilizando evaluación de corto circuito:

- En una expresión a && b, si a evalúa a false, la expresión b no se evalúa.
- En una expresión a || b, si a evalúa a true, la expresión b no se evalúa.

Esta restricción es obligatoria y forma parte de la semántica del lenguaje, ya que permite evitar evaluaciones innecesarias y efectos secundarios no deseados.

```
func esValido(x int32) bool {
    println("Evaluando función")
    return x > 0
}

var flag bool = false

// La función esValido no se ejecuta debido al cortocircuito
flag && esValido(10)
```

3.3.9. Sentencias de control de flujo

If:
La sentencia if permite la ejecución condicional de un bloque de sentencias en función del resultado de una expresión booleana.

```
<ifStmt> ::= <if> <cond> { <stmts> }
           | <if> <cond> { <stmts> } <else> { <stmts> }
```

- La condición debe ser de tipo bool.
- No se permiten paréntesis alrededor de la condición.
- Se permite una inicialización opcional, cuyo alcance se limita al if y sus bloques else.
- else if se modela como un else que contiene otro if.

```
x := 10

if x > 0 {
    println("x es positivo")
} else {
    println("x es cero o negativo")
}
```

Switch:
La sentencia switch selecciona un bloque de ejecución entre múltiples alternativas basadas en el valor de una expresión.

```
<switchStmt> ::= <switch> <exp> {
    <case> <exp>:
        <stmts>
    ...
    <default>:
        <stmts>
```

```
}
```

```
switch day {
case 1:
    fmt.Println("Lunes")
case 2, 3:
    fmt.Println("Martes o Miércoles")
default:
    fmt.Println("Otro día")
}
```

For:
Es la única estructura de iteración del lenguaje, cubriendo los comportamientos de for, while y bucles infinitos.

```
<forStmt> ::= <for> <varDcl>; <cond>; <exp> { <stmts> }
      | <for> <cond> { <stmts> }
      | <for> { <stmts> }
```

- No existe la palabra reservada while.
- La condición debe ser booleana.
- La inicialización puede declarar variables con alcance local al bucle.
- El bucle infinito se controla mediante break o return.

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}

x := 3

for x > 0 {
    fmt.Println(x)
    x--
}

for {
    fmt.Println("Bucle infinito")
    break
}
```

3.3.10. Sentencias de transferencia

Break:
La sentencia break finaliza inmediatamente la ejecución de un bucle o de un switch.

```go
for i := 0; i < 10; i++ {
    if i == 5 {
        break
    }
    fmt.Println(i)
}

// Output esperado:
// 0
// 1
// 2
// 3
// 4
```

Continue:
La sentencia continue interrumpe la iteración actual de un bucle y continúa con la siguiente.

```go
for i := 0; i < 5; i++ {
    if i == 2 {
        continue
    }
    fmt.Println(i)
}

// Output esperado:
// 0
// 1
// 3
// 4
```

Return:
La sentencia return finaliza la ejecución de una función y devuelve opcionalmente un valor.

```go
func suma(a int, b int) int {
    return a + b
}

func main() {
    resultado := suma(3, 4)
    fmt.Println(resultado)
}

// Output esperado:
// 7
```

3.3.11. Arreglos

Un arreglo es una estructura de datos que almacena una colección ordenada y de tamaño fijo de elementos del mismo tipo. El tamaño del arreglo forma parte de su tipo y se determina en tiempo de compilación. Los arreglos se indexan a partir de cero y su acceso se realiza mediante un índice entero. Todos los elementos del arreglo se inicializan automáticamente con el valor por defecto de su tipo si no se proporciona una inicialización explícita.

```
<arrayDcl> ::= <var> <id> [ <exp> ] ... <type>
             |  <var> <id> [ <exp> ] ... <type> = [ <exp> ]
                <type> { <exp>, ... }
```

Simple:

```
var a [5]int
```

Inicializada:

```
var b [3]int = [3]int{1, 2, 3}
```

Asignación:

```
var nums [4]int

nums[0] = 10
nums[1] = 20

fmt.Println(nums[0])
fmt.Println(nums[1])

// Output esperado:
// 10
// 20
```

Arreglo de distintos tipos:

```
var letters [3]rune = [3]rune{'a', 'b', 'c'}
var names [2]string = [2]string{"Ana", "Luis"}
```

Multidimensionalidad:

```
var m [2][3]int
```

```
var mat [2][2]int = [2][2]int{
    {1, 2},
    {3, 4},
}

var grid [2][3]int

grid[0][1] = 5
grid[1][2] = 8

fmt.Println(grid[0][1])
fmt.Println(grid[1][2])

// Output esperado:
// 5
// 8
```

For con arreglos:

```
var m [2][2]int = [2][2]int{
    {1, 2},
    {3, 4},
}

for i := 0; i < 2; i++ {
    for j := 0; j < 2; j++ {
        fmt.Println(m[i][j])
    }
}

// Output esperado:
// 1
// 2
// 3
// 4
```

3.3.12. Funciones y parámetros

Una función es una unidad de código reutilizable que encapsula un conjunto de sentencias y puede recibir cero o más parámetros de entrada. Cada parámetro está asociado a un tipo estático, determinado en tiempo de compilación. Las funciones pueden devolver un valor de retorno, el cual su tipo debe declararse explícitamente.

Los parámetros se pasan por valor, siguiendo el comportamiento de Go. No obstante, mediante el uso de punteros, una función puede modificar el estado de una variable externa. El alcance de los parámetros y variables locales está limitado al cuerpo de la función.

```
<functionDcl> ::= <func> <id> ( <params> ) <type> { <stmts> }
                | <func> <id> ( <params> ) ( <type>, ...) { <stmts> }
```

```go
func suma(a int, b int) int {
    return a + b
}

func main() {
    r := suma(3, 4)
    fmt.Println(r)
}
// Output esperado:
// 7
```

Múltiples retornos:
Una función puede devolver más de un valor en una sola invocación. Cada valor de retorno tiene un tipo definido explícitamente en la firma de la función. Los valores retornados se evalúan y devuelven simultáneamente.

Esta característica elimina la necesidad de estructuras auxiliares o excepciones para comunicar estados adicionales, favoreciendo un estilo de programación claro y explícito.

```go
func dividir(a int, b int) (int, bool) {
    if b == 0 {
        return 0, false
    }
    return a / b, true
}
func main() {
    resultado, ok := dividir(10, 2)
    if ok {
        fmt.Println(resultado)
    }
}
// Output esperado:
// 5
```

Parámetros por referencia mediante punteros
Por defecto, los parámetros de una función se pasan por valor, lo que implica que la función recibe una copia del argumento original y cualquier modificación realizada sobre el parámetro no afecta a la variable externa. Este comportamiento es consistente con el modelo de paso de parámetros de Go y con la semántica general del lenguaje Golampi.

No obstante, el lenguaje permite el paso de parámetros por referencia mediante el uso de punteros. Un parámetro declarado como puntero recibe la dirección de memoria de la variable original, permitiendo que la función modifique directamente su contenido. Para obtener la dirección de una variable se utiliza el operador de referencia (&), y para acceder o modificar el valor apuntado se emplea el operador de desreferenciación (*).

Este mecanismo resulta especialmente útil para funciones que deben modificar estructuras de datos complejas, como arreglos, sin necesidad de retornarlas explícitamente.

```
// Ordenamiento SIN punteros (paso por valor)
// La función recibe una copia del arreglo
func sort(a [5]int) [5]int {
    for i := 0; i < 5; i++ {
        for j := 0; j < 4; j++ {
            if a[j] > a[j+1] {
                temp := a[j]
                a[j] = a[j+1]
                a[j+1] = temp
            }
        }
    }
    return a
}
// Ordenamiento CON punteros (paso por referencia)
// La función recibe la dirección del arreglo original
func sortRef(a *[5]int) {
    for i := 0; i < 5; i++ {
        for j := 0; j < 4; j++ {
            if a[j] > a[j+1] {
                temp := a[j]
                a[j] = a[j+1]
                a[j+1] = temp
            }
        }
    }
}

func main() {
    nums1 := [5]int{5, 3, 4, 1, 2}
    nums2 := [5]int{5, 3, 4, 1, 2}

    // Aquí es obligatorio re-asignar, ya que sort trabaja sobre una
copia
    nums1 = sort(nums1)

    // Aquí no se re-asigna, porque sortRef modifica el arreglo original
```

```
    sortRef(&nums2)

    fmt.Println(nums1[0], nums1[1], nums1[2], nums1[3], nums1[4])
    fmt.Println(nums2[0], nums2[1], nums2[2], nums2[3], nums2[4])
}

// Output esperado:
// 1 2 3 4 5
// 1 2 3 4 5
```

3.3.13. Funciones embebidas

El lenguaje incluye un conjunto reducido de funciones embebidas (built-in) que están disponibles de forma global y no requieren una declaración previa por parte del usuario. Estas funciones proveen operaciones básicas de entrada/salida, conversión y manipulación elemental de datos.

Las funciones embebidas forman parte del entorno del lenguaje y se resuelven directamente durante el análisis semántico o la generación de código, sin necesidad de una definición explícita en el programa fuente.

Función	Definición
`fmt.Println`	Imprime uno o más valores en la salida estándar, separados por espacios y seguidos de un salto de línea.
`len`	Retorna la longitud de una cadena de texto o de un arreglo. El valor retornado es de tipo entero.
`now`	Retorna la fecha y hora actual del sistema en formato de cadena (`YYYY-MM-DD HH:MM:SS`).
`substr`	Extrae una subcadena a partir de una cadena dada, indicando el índice inicial y la longitud deseada. Genera error si los índices son inválidos.
`typeOf()`	Retorna el tipo de una variable en un string. `(int, float32, rune, etc)`

fmt.Println():

```
fmt.Println("Resultado:", 10, true)
```

```
// Output esperado:
// Resultado: 10 true
```

len():

```
var a [4]int
var s string = "Hola"

fmt.Println(len(a))
fmt.Println(len(s))

// Output esperado:
// 4
// 4
```

now():

```
var fecha string = now()
fmt.Println(fecha)

// Output esperado (ejemplo):
// 2026-01-21 10:30:00
```

substr():

```
var s string = "Compiladores"
var sub string = substr(s, 0, 4)

fmt.Println(sub)

// Output esperado:
// Comp
```

typeOf():

```
func main() {
    x := 34
    fmt.Println(typeOf(r))
    // resultado
    // int32
}
```

3.3.14. Función main

La función main es el punto de entrada del programa. Su ejecución inicia automáticamente al comenzar la ejecución del programa y no puede ser invocada explícitamente por el usuario. Todo programa válido debe definir exactamente una función main.

La función main no recibe parámetros y no retorna valores. Su finalidad es coordinar la ejecución del programa mediante llamadas a otras funciones y la ejecución de sentencias de control.

Hoisting:

El lenguaje soporta hoisting de funciones, lo que implica que todas las funciones se consideran declaradas antes de la ejecución del programa, independientemente del orden en el que aparezcan en el código fuente.

Esto permite que una función sea llamada antes de su definición textual, siempre que su firma sea válida.

```
func main() {
    saludar()
}

func saludar() {
    fmt.Println("Hola")
}
```

- Debe existir una y solo una función main.
- No puede recibir parámetros.
- No puede retornar valores.
- No puede ser llamada explícitamente desde el código.
- Es la primera función ejecutada por el runtime.

3.4. Reportes

3.4.1. Reporte de errores

El intérprete debe ser capaz de identificar y registrar los principales errores que puedan presentarse en las fases léxica, sintáctica y semántica sin detener el análisis al encontrar el primer fallo.
En la fase léxica deberá detectar errores tales como:

- Símbolos o caracteres no pertenecientes al lenguaje.

En la fase sintáctica deberá identificar errores estructurales fundamentales, es decir, que impidan la creación del árbol sintáctico, tales como:

- Construcciones incompletas evidentes.

En la fase semántica se deben reportar errores como:

- Uso de variables no declaradas.
- Reutilización de identificadores en el mismo ámbito.
- Incompatibilidad de tipos en operaciones o asignaciones.

El reporte de errores debe presentarse como una tabla, indicando claramente el tipo de error, su ubicación y una descripción comprensible.

#	Tipo	Descripción	Línea	Columna
1	Léxico	Símbolo no reconocido: @	12	8
2	Sintáctico	Se esperaba ';' después de la instrucción	25	14
3	Semántico	Variable 'x' no declarada en el ámbito actual	40	5
4	Semántico	Identificador ‘y’ ya ha sido declarado	45	2
5	Semático	Operación ‘suma’ inválida entre ‘int’ y ‘string’	50	13

3.4.2. Reporte tabla de símbolos

La tabla de símbolos representa el resultado del análisis semántico. Debe contener todos los identificadores declarados durante la ejecución del programa, indicando su tipo, valor (si aplica), ámbito y ubicación en el código. El ámbito es un elemento clave (global, función, bloque, ciclo), ya que demuestra el manejo correcto de contextos.

Este reporte debe ser tabular y reflejar el estado final de la ejecución o análisis.

Identificador	Tipo	Ámbito	Valor	Línea	Columna
bubbleSort	función	global	—	1	1
arr	arreglo	bubbleSort	—	1	18
n	entero	bubbleSort	—	1	23
i	entero	bubbleSort	0	5	9
j	entero	bubbleSort	0	6	9
temp	entero	bubbleSort	—	7	9
string1	cadena	global	"123"	27	5
string1	arreglo	global	{5,3,8,1,2}	28	5

3.5. Alcance del proyecto

Para que los estudiantes sepan hasta dónde deben llegar para una correcta calificación se establecen los siguientes alcances.

Alcance obligatorio (Funcionalidades esenciales)

- Editor de código: La aplicación debe permitir crear, abrir, editar y guardar archivos.
- Análisis léxico y sintáctico para el lenguaje de entrada.
- Análisis semántico: validación de tipos, declaración y alcance (scope) de identificadores, manejo de entornos.
- Ejecución: el intérprete localiza la función principal y la ejecuta.
- Se debe agregar al auxiliar al repositorio de Github/GitLab:
    - Sección A: DiegoCali
    - Sección B: rubenralda
    - Sección N: AllVides

Realizar todas las operaciones aritméticas, lógicas, declaración de variables con todos los tipos, función de imprimir en consola, ya que será lo mínimo en todos los archivos de entrada de calificación que se va a utilizar.

3.6. Entregables

Tipo	Descripción
Prototipo	Desarrollo de una versión funcional inicial del proyecto.
Documentación Técnica	Documento en Markdown que describa: ● Gramática formal de Golampi ● Diagrama de clases y de flujo de procesamiento (tabla de símbolos).
Código Fuente	Repositorio Git organizado (GitHub) con todo el código.
Manual de Usuario	- Manual paso a paso para instalar y usar la herramienta - Instrucciones para crear, editar y ejecutar código, así como para interpretar los reportes (tabla de símbolos y errores).

	- Capturas de pantalla de la interfaz y ejemplos de sesión de uso.

4. Material de apoyo

La documentación de la herramienta para generar el análisis sintáctico se encuentra en este link https://github.com/antlr/antlr4.

5. Metodología

1. **Investigación preliminar**: Revisar conceptos de teoría de compiladores como el análisis léxico, sintáctico y semántico. Estudiar ejemplos básicos de gramática. Configurar el entorno para ejecutar el programa y repositorio Git.
2. **Diseño del sistema**: Definir la gramática formal, cubriendo variables, estructuras de control, arreglos, funciones etc. Documentar la estructura de carpetas y el patrón arquitectónico a utilizar o flujo del desarrollo.
3. **Desarrollo**:
    a. **Bloque 1**: Implementar y probar el analizador léxico.
    b. **Bloque 2**: Generación del analizador sintáctico mediante ANTLRv4.
    c. **Bloque 3**: Implementar el análisis semántico (tabla de símbolos, validaciones de tipos). Crear casos de prueba para semántica y manejo de errores.
    d. **Bloque 4**: Desarrollar la GUI. Integrar el intérprete con la interfaz.
4. **Pruebas y ajustes**: Se realizarán pruebas para verificar que la aplicación funciona correctamente en diferentes condiciones.

6. Recursos y herramientas a utilizar

Los estudiantes deberán emplear las siguientes tecnologías y herramientas en el desarrollo:

- Lenguaje de entrada: Golang
- Lenguaje de implementación: PHP
- Generación de analizadores léxico y sintáctico: ANTLRv4.
- Interfaz gráfica de usuario: HTML y CSS como base.

7. Desarrollo de Habilidades Blandas

En los proyectos individuales, cada estudiante asume plenamente la planificación, ejecución y entrega, promoviendo no solo habilidades técnicas sino también Profesionales y de aprendizaje continuo.

6.1.1 Gestión del tiempo
Planificar tareas y establecer hitos propios para asegurar el avance constante y la entrega puntual de cada fase del intérprete.

6.1.2 Autonomía en la investigación
Buscar y seleccionar recursos, ejemplos de gramáticas y buenas prácticas para resolver dudas y guiar el diseño sin depender de supervisión constante.

6.1.3 Resolución creativa de problemas
Enfrentar errores en la gramática, el parser o la lógica de ejecución con iniciativas propias: diseñar casos de prueba, depurar paso a paso y ajustar la implementación.

6.1.4 Calidad de la documentación y pruebas
Mantener un README claro y documentar adecuadamente los módulos del sistema (léxico, sintáctico, semántico e interfaz gráfica), facilitando el mantenimiento, la comprensión del proyecto y su evaluación, junto con evidencia de pruebas funcionales.

8. Cronograma

El cronograma describe las etapas clave del proyecto, los plazos estimados para cada una, y el proceso de asignación, elaboración y calificación de las tareas. Los estudiantes deberán seguir este plan para asegurar que el proyecto avance de manera organizada y cumpla con los plazos establecidos. Cada fase incluye la asignación de tareas, el tiempo estimado para su elaboración, y el momento de su calificación.

Tipo	Fecha Inicio	Fecha Fin
Elaboración	2/02/2026	12/03/2026
Calificación	13/03/2026	28/03/2026